3.4a. Comprender la representación de un número en el sistema de numeración posicional es esencial para solucionar el problema. El haber trabajado el ejercicio 2.21 puede ayudarte a ello.

3.5a. Haber trabajado con el ejercicio 3.4 puede ser de gran ayuda para saber cómo *obtener* los dígitos que componen un número.

3.7a. Puede ser muy útil definir una función aleatoria que reciba dos enteros como parámetros de entrada, y devuelva un número entero aleatorio entre dichos parámetros. Con el ejercicio 3.13 puedes aprender a definir variables aleatorias.

3.7b. Aquí recogemos algunas expresiones que permiten determinar si un punto (x, y), en dos dimensiones, o (x, y, z), en tres, es interior a las figuras geométricas que se describen.

- Círculo con centro en el punto (a, b) y radio r:
$$(x-a)^2 + (y-b)^2 \leq r^2$$
- Elipse con centro en el punto (a, b) y radio de la abcisa r_1 y radio de la ordenada r_2:
$$\frac{(x-a)^2}{{r_1}^2} + \frac{(y-b)^2}{{r_2}^2} \leq 1$$
- Esfera con centro en el punto (a, b, c) y radio r:
$$(x-a)^2 + (y-b)^2 + (z-c)^2 \leq r^2$$
- Cilindro con base en el círculo de centro (a, b, c), radio r y altura h:
$$(x-a)^2 + (y-b)^2 = r^2$$
$$c \leq z \leq c+h$$

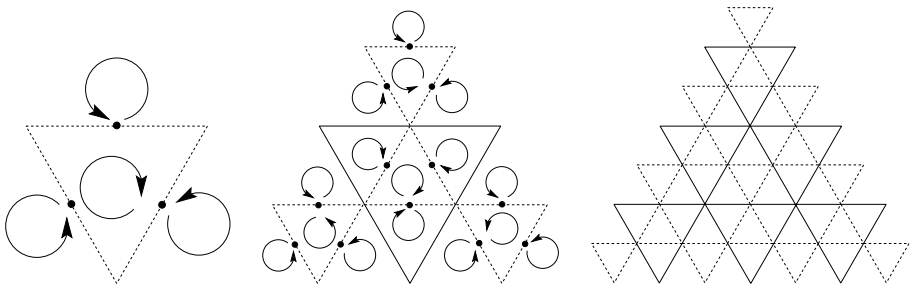
3.15a. En el ejercicio 3.5 tienes funciones que te serán útiles.

3.16a. El ejercicio 3.13 te será útil para definir las funciones aleatorias que necesitas utilizar en la simulación de los juegos que propone el ejercicio.

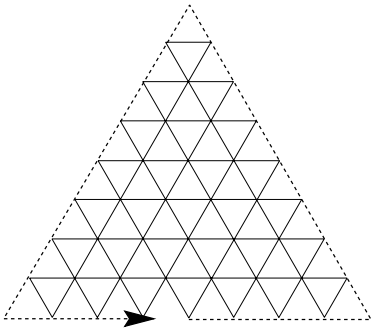
3.21a. C++ tiene recursión.

3.25a. El nivel más básico consiste en dibujar un triángulo, que contiene otros tres triángulos, de lado la mitad del exterior; este proceso podemos repetirlo tantas veces como deseemos dentro de cada uno de los triángulos resultantes del nivel anterior.

Lo mejor es escribir un subprograma recursivo que dibuje estos triángulos anidados. En realidad el programa recursivo deberá dibujar solamente el interior de los triángulos:



Después, se dibujará el borde del triángulo principal,



con lo que habremos terminado el dibujo que nos proponíamos.

3.26a. Si no tienes suficiente práctica con el álgebra matricial, la fórmula que define L_n te puede resultar extraña. Cuando se construye una matriz juntando otras, se entiende que las internas pierden el caparazón que sujeta sus elementos y éstos pasan a formar parte de la matriz externa. Por eso,

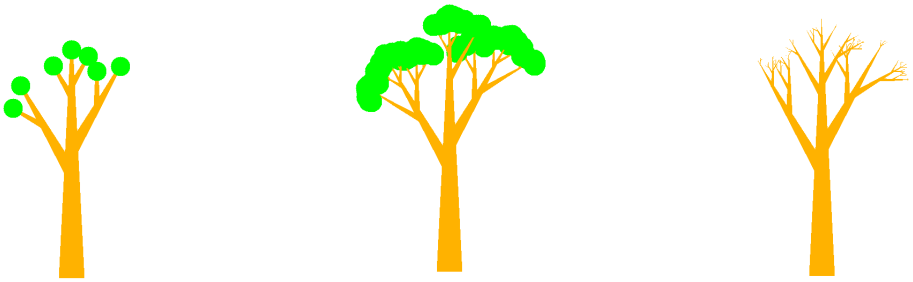
$$L_1 = \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix}$$

y

$$L_2 = \begin{bmatrix} 0 & 12 & 3 & 15 \\ 8 & 4 & 11 & 7 \\ 2 & 14 & 1 & 13 \\ 10 & 6 & 9 & 5 \end{bmatrix}.$$

3.27a. Para obtener un mayor realismo en el dibujo conviene tener en cuenta lo siguiente:

- Para obtener un árbol con suficiente detalle de ramificación hay que iterar el proceso unos ocho niveles.
- Si se llega hasta niveles altos, las hojas pueden ser demasiado pequeñas para ser apreciadas, tendríamos en este caso un árbol típicamente invernal, sin hojas. Si queremos ver un árbol con las hojas verdes propias de la primavera, la solución consiste en establecer un radio mínimo para los discos que acaban siendo las hojas. Dependiendo de ese radio mínimo el árbol aparecerá más o menos verde.
- Siguiendo el proceso tal y como se ha dicho obtenemos un árbol demasiado *geométrico*; para obtener más realismo se puede introducir algo de aleatoriedad: los subárboles más pequeños se pintarán dependiendo de un cierto número escogido de forma aleatoria.
- Por último se pueden intentar diversas configuraciones para obtener distintos tipos de árboles.



31 Incierta igualdad de números reales

103

Una primera idea consiste en cambiar la comparación $x = y$ por la siguiente,

$$|x - y| < \varepsilon$$

siendo ε el margen de tolerancia entre dos reales *supuestamente iguales:*

```
bool proximos(double const x, double const y) {
  float const eps = 1E-6;
  return fabs(x-y) < eps;
}
```

Pero esta solución presenta el inconveniente de conceder el mismo margen de tolerancia a cualquier par de reales, independientemente de su orden de magnitud. Por ejemplo, un margen $\varepsilon = 10^{-3}$ resulta excesivo para comparar cifras de órdenes pequeños (tales como medidas de átomos), y a la vez ese mismo margen es en cambio estrechísimo para números de órdenes grandes (como son las distancias astronómicas).

Una solución consiste en sustituir la expresión $|x - y| < \varepsilon$ por la siguiente,

$$\frac{|x - y|}{x^2 + y^2} < \varepsilon$$

que relativiza la distancia según su orden de magnitud.

Ahora surge la posible división por cero, que se presenta en el caso de ser x e y nulos o de valor muy pequeño. Por ello, en vez de la expresión anterior, optamos por la siguiente:

$$|x - y| < \varepsilon(x^2 + y^2).$$

En resumen, basta con usar la función anterior, cambiando su última línea por ésta:

```
return fabs(x-y) < eps * (x*x + y*y);
```

33 Ponle tú el título

103

El resultado es la siguiente matriz:

0	1	2	3	4	5
5	0	1	2	3	4
4	5	0	1	2	3
3	4	5	0	1	2
2	3	4	5	0	1
1	2	3	4	5	0

36 Cuentaletras

105

Podría parecer que una solución a este problema requiere un par de bucles anidados: el interno para contar las letras de cada palabra; el externo para recorrer cada palabra de un texto. Pero esta primera intuición lleva a una solución muy compleja. Para que se entienda por qué, nos dejaremos llevar por un momento al fracaso para, seguidamente, mostrar una solución más limpia.

3.6.1 Bucles anidados

Supongamos que tenemos una función predicado llamada `esLetra` que determina si un carácter es una letra. El siguiente bucle cuenta las letras consecutivas que encuentra en el fichero `in` y, por ende, si se ejecuta al principio de una palabra, cuenta las letras que tiene.

```
int letras = 0;
char siguiente;
in.get(siguiente);
while (esLetra(siguiente)) {
  letras++;
  in.get(siguiente);
}
```

Pero este bucle ignora un detalle fundamental siempre que se trabaje con ficheros: antes de leer, se ha de comprobar que no hemos llegado al fin.

Centremos nuestra atención en el bucle. Parecerá que el problema se soluciona así:

```
while (esLetra(siguiente) && !in.eof()) {
  letras++;
  in.get(siguiente);
}
```

Ahora el problema está en el otro extremo porque comprobamos demasiado pronto si hay algo que leer. De esta forma, si el fichero acaba con una letra, no la tendremos en cuenta y, por tanto, atribuiremos un carácter menos a la última palabra.

El momento oportuno para comprobar si podemos leer un nuevo carácter es justamente antes de leerlo:

```
while (esLetra(siguiente)) {
  letras++;
  if (!in.eof()) in.get(siguiente);
  else ⟨¿Qué hacer en este caso?⟩;
}
```

Pero ¿qué podemos poner en `siguiente` si no queda nada por leer del fichero? Se suele recurrir a un valor que termine el bucle y no tenga ningún efecto sobre la cuenta; por ejemplo, un espacio.

Ahora que hemos dado con el bucle interno que cuenta las letras de una palabra, sólo queda rodearlo de uno que lo repite hasta que se acabe el fichero. Por supuesto, entre palabra y palabra, hay espacios que es necesario quitar con otro pequeño bucle.

```
while (!in.eof()) {
  char siguiente;
  in.get(siguiente);
  while (!esLetra(siguiente) && !in.eof()) in.get(siguiente);
  int letras = 0;
  while (esLetra(siguiente)) {
    letras++;
    if (!in.eof()) in.get(siguiente);
    else siguiente = ' ';
  }
  if (letras > 0) out << letras;
}
```

3.6.2 Un solo bucle

Hay algunas otras formas de organizar los bucles doblemente anidados anteriores; pero no conocemos ninguna que sea suficientemente elegante. Busquemos una vía para escapar de este fracaso.

Lo que intuitivamente puede parecer un par de bucles anidados, muchas veces se puede aplanar a uno solo si recurrimos al concepto de *estado*. En nuestro problema, cuando estamos recorriendo un fichero, tenemos dos posibles estados: o bien estamos dentro de una palabra, o bien estamos fuera. Estaremos *dentro de una palabra* cuando se haya leído alguna de sus letras, pero todavía no se haya leído un carácter que indique que se ha acabado. Estaremos fuera en cualquier otro caso.

Hay que analizar cuatro casos, resultado de combinar dos posibilidades sobre el carácter (si es o no letra) con dos estados (estamos dentro o fuera de una palabra). El siguiente código es un boceto de lo que hay que hacer en cada caso. Al igual que en la solución anterior, usamos la variable entera `letras` para contar las letras de una palabra.

```
if (esLetra(siguiente)) {
  if (⟨estamos dentro de una palabra⟩) {
    letras++;
  } else {
    letras = 1;
    ⟨Se anota que estamos dentro de una palabra⟩
  }
} else if (⟨estamos dentro de una palabra⟩) {
  out << letras;
  ⟨Se anota que estamos fuera de una palabra⟩
}
```

El estado se suele representar con diferentes técnicas. Algunas son explícitas; por ejemplo, cuando se utiliza una variable que toma un valor distinto por cada posible estado. Aquí utilizaremos un técnica implícita; aunque en un principio, el propósito de la variable `letras` era contar el número de letras de cada palabra, también servirá para indicar el estado, porque ocurre que estamos en una palabra precisamente si `letras > 0`.

```
int letras = 0;
while (!in.eof()) {
  char siguiente;
  in.get(siguiente);
  if (esLetra(siguiente)) {
    letras++;
  } else if (letras > 0) {
    out << letras;
    letras = 0;
  }
}
if (letras > 0) out << letras;
```

3.6.3 Ser letra

Para implementar el predicado `esLetra(siguiente)` podemos aprovechar la numeración correlativa de los caracteres alfabéticos en el código ASCII,

```
bool esLetra(char const caracter) {
  return ('a' <= caracter) && (caracter <= 'z')
         || ('A' <= caracter) && (caracter <= 'Z');
}
```

o, directamente, podemos utilizar la función `isalpha()`, que se encuentra en `ctype.h`:

```
bool esLetra(char const caracter) {
  return isalpha(caracter);
}
```

Aunque esta solución ignora la letra `ñ` y las tildes, no será difícil modificarla para que se tengan en cuenta.

3.11 Sucesión bicicleta

109

Periodo La idea es sencilla: considerando los términos consecutivos c_1 y c_2 (con valores iniciales b_1 y b_2), basta con hacerlos avanzar hasta que se tenga, nuevamente, $c_1 = b_1$ y $c_2 = b_2$, y el período p será el número de avances efectuados:

```
int periodo(double const primero, double const segundo) {
  double const b1 = primero, b2 = segundo;
  double c1 = b1, c2 = b2;
  int numPasos = 0;
  do {
    double c = (c2+1)/c1; c1 = c2; c2 = c;
    numPasos++;
  } while (b1 != c1 || b2 != c2);
  return numPasos;
}
```

Teniendo en cuenta que el bucle con que generamos los términos b_i de la sucesión está controlado por la expresión siguiente,

$$(c_1 = b_1) \wedge (c_2 = b_2)$$

que valora la igualdad de números reales, es preciso evitar el peligro de caer en un bucle infinito debido a las deficiencias de precisión en la representación de los números reales. Por ello, usamos la función `proximos`, que se explica en el ejercicio 3.1. Y así sustituimos la condición `b1 != c1 || b2 != c2`, de salida del bucle por esta otra:

```
!proximos(b1, c1) || !proximos (b2, c2)
```

Serie Ahora, el modo trivial sería avanzar a través de la sucesión, pasando por todos los términos que se desea sumar, y añadiéndolos uno a uno. Pero al ser cíclica la serie, es más eficiente proceder como sigue:

1. Se calcula primero la suma `sumCiclo` de los términos de un ciclo:

   $$\texttt{sumaCiclo} = \sum_{i=1}^{p} b_i$$

   Este cálculo se puede llevar a cabo a la vez que se halla p, como se ha hecho en el apartado anterior.

2. Se averigua cuántos ciclos completos (`numCiclos`) hay en n términos y cuántos términos sueltos (`numSueltos`) restan hasta el n-ésimo. Se multiplican la suma anterior y `numCiclos`.

3. Se suman los términos sueltos ignorados en los `numCiclos` completos.

Para solucionar el primer punto adaptamos la función del apartado anterior para que calcule, además del período, la suma de los términos de un ciclo. Puesto que se devuelven dos valores, la función se convierte en el procedimiento `recorrerCiclo`. Lo único que hay que cambiar con respecto a la función anterior es la cabecera,

```
void recorrerCiclo(double const primero, double const segundo,
                   double& sumaCiclo, int& periodo)
```

y además, en cada vuelta del bucle será necesario acumular el valor actual calculado:

```
sumaCiclo = sumaCiclo + b;
```

El valor inicial de `sumaCiclo` es 0.

Así la suma de los N primeros términos de la sucesión se calcula con la siguiente función:

```
double sumaN(double const primero, double const segundo, int const hasta) {
  double sumaCiclo;
  int peri;
  recorrerCiclo(primero, segundo, sumaCiclo, peri);
  int numCiclos = hasta / peri;
  int numSueltos = hasta % peri;
  sumaCiclo = sumaCiclo * numCiclos;
  double c1 = primero, c2 = segundo;
  for (int i = 0; i < numSueltos; i++) {
    sumaCiclo = sumaCiclo + c1;
    double b = (c2+1) / c1; c1 = c2; c2 = b;
  }
  return sumaCiclo;
}
```

3 19 Número de ceros en que termina un factorial

118

Grado de multiplicidad El grado de multiplicidad g de d en n es la máxima potencia del factor d que es divisor de n; o sea: $n = k \times d^g$, donde k no es divisible por d. El grado de multiplicidad se puede calcular fácilmente, efectuando las divisiones enteras correspondientes y llevando a la vez la cuenta del número de ellas efectuadas:

$$24 \xrightarrow{/2} 12 \xrightarrow{/2} 6 \xrightarrow{/2} 3$$

Es decir:

```
int grado(int const d, int const n) {
  int auxN = n;
  int grado = 0;
  while (auxN % d == 0) {
    auxN = auxN / d;
    grado = grado++;
  }
  return grado;
}
```

Se ha de exigir que $n \geq 1$ y que $d \geq 2$ para que esté definido el grado de multiplicidad.

Ídem, recursiva El caso base es, trivialmente, el de multiplicidad cero, cuando el primer entero no es múltiplo del segundo,

$$\texttt{grado(7, 24)} \rightsquigarrow 0$$

y el caso recurrente se halla a partir del grado de multiplicidad del primero, dividido exactamente por el segundo:

$$\texttt{grado(2, 24)} \rightsquigarrow 1 + \texttt{grado(2, 12)}$$

Es decir:

```
int gradoRec(int const d, int const n) {
  if ((n % d) != 0) {
    return 0;
  } else {  // n es múltiplo de d
    return 1 + gradoRec(d, n/d);
  }
}
```

Multiplicidad de 5 Este apartado no es más que una aplicación directa de la función `grado`, definida en el caso anterior:

```
int grado5(int const n) {
  return grado(5, n);
}
```

Los ceros de un factorial, al fin Cada cero en un producto requiere la existencia de un 2 y un 5 entre los factores de dicho producto. Como en $1 \times 2 \times 3 \times 4 \ldots$ siempre hay más doses que cincos, el número de cincos que haya entre los factores determina el número de ceros que habrá en el producto. Así pues, en $n!$ habrá tantos ceros como cincos haya en las descomposiciones de los factores $1, 2, 3, \ldots, n$.

```
int ceros(int const n) {
  int tot = 0;
  for (int i = 1; i <= n; i++) {
    tot = tot + grado5(i);
  }
  return tot;
}
```

3.21 ¿Cuál es el mejor orden para recibir los datos de un polinomio?

119 129

La principal característica de este problema es que el número de coeficientes del polinomio es un dato implícito; en ningún momento el usuario del programa dará el valor n; es responsabilidad de nuestro código calcularlo a partir del número de datos, o mejor, ser capaz de operar sin llegar a conocerlo jamás.

Vamos a darle un nombre de letra griega a cada orden propuesto: el del empresario será δ, el nuestro α, el intermedio será β, mientras que γ será un cuarto orden $a_0, \ldots, a_n, b$.

Tanto el orden α como el β son muy sencillos. En ambos casos vamos a guardar en la variable `b` el punto de evaluación; por la variable `a` irán pasando los sucesivos coeficientes, y por `r` irán pasando los resultados parciales de la evaluación. Para el orden α necesitamos las sucesivas potencias de `b`. Se pueden conseguir de dos formas: calculándolas completamente en cada momento, o de forma incremental en una variable auxiliar que aquí llamaremos `p`:

```
void eval(int& r) {
  r = 0;
  int p = 1;
  int b;
  cin >> b;
  while (!eoln(cin)) {
    int a;  cin >> a;
    r = r + a*p;
    p = p * b;
  }
}
```

El orden β se resuelve fácilmente con la pista 2.8a. Aunque el código es más sutil, también es más breve. Las ideas fundamentales son las mismas que para convertir una secuencia de dígitos en el número que denotan:

```
void eval(int& r) {
  r = 0;
  int b;  cin >> b;
  while (!eoln(cin)) {
    int a;  cin >> a;
    r = r*b + a;
  }
}
```

Los dos órdenes que quedan por resolver son más complicados, pero se resuelven fácilmente haciendo uso adecuado de la recursión. Empezemos con γ. Cuando nuestro procedimiento se enfrente a la secuencia $a_0, \ldots, a_n, b$, aparentemente no puede hacer nada útil: para poder evaluar el polinomio tenemos que multiplicar a cada a_i por una potencia adecuada de b; pero no conoceremos b si no hemos leído (y perdido) antes todos los a_i. Hace falta un cambio de enfoque. Pensemos recursivamente. Si leemos a_0 resultará que la secuencia se ha convertido en $a_1, \ldots, a_n, b$; es un elemento más corta y, por tanto, tiene sentido intentar resolver este subproblema de forma recursiva. Supongamos que lo hemos hecho así y llamemos r_1 al resultado. Para construir la solución del problema original basta con calcular $a_0 + r_1 b$. De esto se deduce que nuestro procedimiento recursivo, además de devolver el valor de la evaluación del subproblema, también tendrá que devolver el valor de b. El caso base de esta recursión es trivial, y se explica mejor con el propio código:

```
void eval(int& r, int& b) {
  int a;  cin >> a;
  if (eoln(cin)) {
    r = 0;
    b = a;
  } else {
    eval(r, b);
    r = a + r*b;
  }
}
```

El orden δ es ligeramente más complejo. La secuencia original es $a_n, \ldots, a_0, b$; el subproblema que resuelve la llamada recursiva es $a_{n-1}, \ldots, a_0, b$. Supongamos, como antes, que devolvemos tanto la evaluación del subproblema (en r) como el punto de evaluación b. Para reconstruir el valor del problema

original necesitamos calcular $a_n b^n + r$. Es decir, es necesario devolver un dato más, que puede ser bien n o bien b^n. En el procedimiento que sigue, hemos optado por devolver b^n en el parámetro `p`:

```
void eval(int& r, int& p, int& b) {
  int a;  cin >> a;
  if (eoln(cin)) {
    r = 0;
    p = 1;
    b = a;
  } else {
    eval(r, p, b);
    r = a*p + r;
    p = p * b;
  }
}
```

3.24 Codificaciones de plantas con cadenas

121

Antes de empezar con la solución de cada uno de los programas conviene definir una constante por cada letra del abecedario de las plantas:

```
char const brote = 'B';
char const tramo = 'T';
char const hojaIzq = 'H';
char const hojaDer = 'A';
char const subramaIzq = 'I';
char const subramaDer = 'D';
char const corte = 'C';
```

Empezamos por realizar una función `crece`, que recibe como argumento una planta (dada simplemente con un `string`) y el número de tramos que queremos añadir a cada brote, y devuelve como resultado la planta resultante.

Esta función es bastante sencilla: basta con recorrer la planta copiándola en una nueva; cuando encontramos un brote, añadimos antes del mismo tantos tramos como indique el parámetro correspondiente:

```
string crece(string const original, int const n) {
  string nuevo = "";
  for (int i = 0; i < original.size(); i++) {
    if (original[i] == brote) {
      for (int j = 0; j < n; j++) nuevo = nuevo+tramo;
    }
    nuevo = nuevo+original[i];
  }
  return nuevo;
}
```

El resto de los programas se pueden realizar de forma sencilla utilizando la recursión. Para el programa `es`, realizaremos una función que crea el níspero con el nivel deseado. Vamos por partes:

- El caso base de esta recursión se tiene cuando `nivel == 0`, en cuyo caso realizaremos un níspero con un solo tronco y varias hojas.

- En el caso recursivo, se realizan varios tramos que dependerán del nivel en el que estemos dibujando la rama y varias subramas. Su distribución concreta se puede cambiar a gusto de Lindomayo.

Es decir:

```
string es(int const nivel) {
  string resultado="";
  if (nivel == 0) {
    resultado = resultado + tramo + hojaIzq +
                tramo + hojaDer + tramo + brote;
  } else {
    ⟨Añadir un trozo de tronco⟩
    resultado = resultado + subramaIzq + es(nivel-1);
    ⟨Añadir un trozo de tronco⟩
    resultado = resultado + subramaDer + es(nivel-1);
    resultado = resultado + es(nivel-1);
  }
  return resultado;
}
```

Para ⟨*Añadir un trozo de tronco*⟩ realizamos un procedimiento `anyadeTramo` que añade tramos según el nivel en que nos encontremos:

```
void anyadeTramo(string& arbol, int const nivel) {
  for (int i = 0; i < nivel; i++) arbol = arbol + tramo;
}
```

Los programas recursivos que faltan se complican algo más, porque en ellos no es posible distinguir *a priori* el caso base de los casos recurrentes.

Empecemos con el programa `nivel`. En líneas generales, este programa recorrerá el árbol, y cuando se encuentre una bifurcación realizará una llamada recursiva para calcular el nivel de esa rama: si la rama tiene un nivel mayor que el encontrado hasta el momento, debemos acordarnos de ella, y como parte del nivel siguiente al actual, será necesario sumar uno al nivel obtenido. Para recorrer el árbol necesitamos una variable auxiliar `posicion` para saber en qué posición del `string` nos encontramos.

Puesto que el análisis de la rama cambia esa posición, el parámetro deberá ser de entrada y salida: como parámetro de entrada, indica la posición inicial donde nos encontramos; y como parámetro salida comunica la última posición del subárbol analizado. Debido al uso de un parámetro de entrada y salida, parece conveniente realizar un procedimiento en lugar de una función; además, también deseamos saber cuál es la rama de mayor nivel, lo que requiere un segundo parámetro de salida y, por tanto, se refuerza la necesidad de que el subprograma sea un procedimiento. El perfil de este procedimiento es el siguiente

```
void calculaNivel(string const arbol, int& posicion, int& nivel);
```

y la llamada principal será así:

```
string arbol = ⟨contenido del arbol⟩;
int nivel = 0;
int posicion = 0;
calculaNivel(arbol, posicion, nivel);
```

Y vamos con el contenido de dicho procedimiento. En primer lugar necesitamos una variable auxiliar para almacenar el nivel máximo de las ramas; como el nivel mínimo que tendremos es 0, éste será su valor inicial. Seguidamente, realizaremos un bucle para recorrer toda la subrama; supondremos que toda

subrama acaba bien en un corte bien en un brote, e iremos analizando lo que nos encontramos:

```
int nivelMax = 0;
while (arbol[posicion] != brote && arbol[posicion] != corte) {
  ⟨Análisis de la posicion actual⟩
  posicion++;
}
nivel = nivelMax;
```

En cada posición deberemos analizar si tenemos una bifurcación o no; en caso afirmativo realizamos una llamada recursiva a la subrama. Por hipótesis de inducción nos devolverá la última posición de la subrama en la variable `posicion` y el nivel de la subrama en una variable auxiliar que definiremos a tal efecto; puesto que hemos calculado el nivel de una subrama deberemos aumentar en uno el nivel calculado:

```
if (arbol[posicion] == subramaIzq || arbol[posicion] == subramaDer) {
  posicion++;
  int nivelAux = 0;
  calculaNivel(arbol, posicion, nivelAux);
  nivelAux++;
  if (nivelAux>nivelMax) nivelMax = nivelAux;
}
```

Si pretendemos calcular la rama de mayor nivel, el procedimiento deberá tener un parámetro de salida adicional que sea la rama de mayor nivel. Es necesario llevar la cuenta de cuál es la rama de mayor nivel calculado hasta ese momento y de la rama leída hasta el momento. Para ello llevamos dos variables cuyo valor inicial será la cadena vacía. El procedimiento queda como sigue:

```
void calculaNivel(string const arbol, int& posicion, string& rama, int& nivel) {
  string tronco = "", ramaMax = "";
  int nivelMax = 0;
  while (arbol[posicion] != brote && arbol[posicion] != corte) {
    ⟨Análisis de la posicion actual⟩
    posicion++;
  }
  nivel = nivelMax;
  ⟨Terminar la rama de nivel máximo⟩
}
```

Para terminar la rama de nivel máximo debemos indicar cuál ha sido la rama; para ésta tenemos que distinguir dos casos: que la rama no tenga subramas (`nivel == 0`) o que sí las tenga (`nivel > 0`). En el primero deberemos devolver el tronco calculado hasta el momento concatenado con la finalización (brote o corte), y en el segundo la rama de máximo nivel calculado:

```
if (nivel == 0) rama = tronco + arbol[posicion];
else rama = ramaMax;
```

En cuanto al análisis es necesario complicar un poco el caso de la bifurcación para poder calcular de forma adecuada la rama, y en el caso de que el carácter actual sea un tramo o una hoja añadirlo a la variable `tronco`:

```
if (arbol[posicion] == subramaIzq || arbol[posicion] == subramaDer) {
  char desviacion = arbol[posicion];
  posicion++;
```

```
    string ramaAux = "";
    int nivelAux = 0;
    calculaNivel(arbol, posicion, ramaAux, nivelAux);
    nivelAux++;
    if (nivelAux > nivelMax) {
      nivelMax = nivelAux;
      ramaMax = tronco + desviacion + ramaAux;
    }
  } else if (arbol[posicion] == tramo || arbol[posicion] == hojaDer ||
             arbol[posicion] == hojaIzq) {
    tronco = tronco + arbol[posicion];
  }
```

Pasemos ahora a solucionar el programa `complu`; la idea será similar a la anterior: realizaremos un programa recursivo que irá recorriendo el árbol; llevaremos un parámetro de entrada y salida que nos indicará la posición por la que vamos recorriendo el árbol y un parámetro de salida que será el resultado de hacer la poda al árbol original. He aquí el perfil de este subprograma,

```
void complu(string rama, int& posicion, string& ramaPodada);
```

así como la llamada inicial al mismo:

```
string arbol = ⟨contenido del árbol⟩;
int posicion = 0;
string arbolPodado = "";
complu(arbol, posicion, arbolPodado);
```

El algoritmo, recursivo, será parecido al del cálculo del nivel pero, como veremos, es necesario añadir ciertos detalles. Iremos construyendo la rama podada según vayamos recorriendo el árbol, iremos copiando los tramos y las hojas y podando las subramas según vayan apareciendo hasta que lleguemos al final o a la mitad de la rama. Por último habrá que finalizar la rama de forma correcta, e indicar la última posición que ocupa la rama.

```
ramaPodada = "";
while (rama[posicion] != brote && rama[posicion] != corte
       && ⟨longitud recorrida⟩ < ⟨longitud de total de la rama⟩/2) {
  ramaPodada = ramaPodada + rama[posicion];
  ⟨Análisis del la posicion actual⟩
  posicion++;
}
⟨Terminar la rama⟩
⟨Colocar posicion al final de la rama⟩
```

Para acabar la rama podada es necesario tener en cuenta si la rama actual tiene longitud cero y acaba ya en un brote, y entonces seguiremos respetando que la rama podada acabe en un brote. Podemos comprobar que la rama tiene longitud cero simplemente examinando si `rama[posicion]` es igual o no a `brote`, porque si la longitud es mayor que cero el elemento en `rama[posicion]` será un tramo:

```
if (rama[posicion] != brote) {
  ramaPodada = ramaPodada + corte;
} else {
  ramaPodada = ramaPodada + brote;
}
```

No debemos, por último, olvidar que hemos de colocar la variable `posicion` al final de la rama:

`posicion` = ⟨*última* `posicion` *de la rama*⟩;

En lo anterior, podemos ver que necesitamos una serie de variables auxiliares: para saber la longitud total de la rama introducimos la variable `longitudTotal`; para saber la posición final de la rama usaremos la variable `posicionFinal`; y para saber la longitud recorrida de la rama, la variable `longitudActual`.

El valor inicial de la variable `longitudActual` será cero, y habrá que incrementarla cada vez que nos encontremos un tramo. Para calcular la longitud total de la rama y su posición final realizaremos un programa recursivo, con el siguiente perfil:

```
void longitud(string const& rama, int& posicion, int& longTotal);
```

El parámetro `posicion` será de entrada y salida: de entrada porque indica la posición de inicio de la rama, y de salida porque nos devolverá la última posición de la rama; el parámetro `longTotal` será de salida. Así antes del bucle deberemos incluir las siguientes líneas:

```
int posicionFinal = posicion;
int longitudTotal = 0;
int longitudActual=0;
longitud(rama, posicionFinal, longitudTotal);
```

El procedimiento `longitud` es similar tanto al del cálculo del nivel como al de la poda, por lo que se deja al lector su desarrollo. Con todo esto, el ⟨*Análisis del la* `posicion` *actual*⟩ deberá realizar las llamadas recursivas si la `posicion` actual es una bifuración o aumentar `longitudActual` si se trata de un tramo:

```
if (rama[posicion] == subramaIzq || rama[posicion] == subramaDer) {
  posicion++;
  string subRamaPodada = "";
  complu(rama, posicion, subRamaPodada);
  ramaPodada = ramaPodada + subRamaPodada;
} else if (rama[posicion] == tramo) {
  longitudActual++;
}
```

El programa `beneficio` tiene una versión trivial que consiste en calcular el valor de un árbol, podar el árbol, calcular el valor del árbol podado y restar las dos cantidades obtenidas.

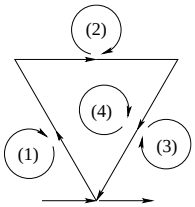
```
int valorEjemplar(string const ejemplar) {
  int resultado = 0;
  for (unsigned int i = 0; i < ejemplar.size(); i++) {
    if (ejemplar[i] == tramo) resultado++;
  }
  return resultado;
}
int beneficio(string const arbol) {
  string arbolPodado;
  int posicion = 0;
  complu(arbol, posicion, arbolPodado);
  return valorEjemplar(arbol) - valorEjemplar(arbolPodado);
}
```

También se puede realizar un algoritmo recursivo similar a los anteriores que calcule el beneficio de podar un árbol sin necesidad de calcular el árbol podado.

Para entender bien el programa es necesario tener en cuenta que en cada momento la plumilla está en una posición y preparada para dibujar en un determinado ángulo.

Ahora, nos centraremos en el dibujo del interior del triángulo, que detallamos así:

> Se dibuja el interior del triángulo que estamos considerando,
> en la posición donde está actualmente la plumilla,
> a la izquierda según la dirección en la que está actualmente la plumilla, y
> al finalizar, el ángulo de la plumilla debe ser el ángulo inicial.



En primer lugar se dibuja una línea desde el lugar *donde estamos* hasta el punto medio del primer lado del triángulo interior, hacemos la primera llamada recursiva (1) y acabamos de dibujar el primer lado. Vamos con el segundo lado: dibujamos la primera mitad, hacemos la segunda llamada recursiva (2) y acabamos de dibujar el segundo lado. Dibujamos la primera mitad del tercer lado, hacemos la tercera llamada recursiva (3), y... en este punto podemos caer en la tentación de completar la mitad que nos falta del tercer lado.

Pero ¡ay! en la descripción que hemos hecho, hemos dibujado los triángulos externos, ¡y nos hemos olvidado del triángulo interior! Este triángulo se puede dibujar en cualquier momento, justo antes o después de dibujar cualquier triángulo externo. Pero ¿cómo? Los triángulos se dibujan siempre en la parte izquierda de la dirección actual, de forma que, para dibujarlo en la parte interna, habrá que girar la plumilla π radianes (180 grados) para que ésta apunte en sentido contrario. Y después de haber hecho la llamada recursiva deberemos volver a la dirección original volviendo a girar π radianes.

Para poder distinguir de forma adecuada cada uno de los niveles de dibujo, cada nivel se dibuja en un color diferente. Así el programa recursivo queda:

```
void dibujarRecursivo(double const lado, int const nivel) {
  if (nivel > 0) {
    ponerColor(nivel);
    entrar();
    dibujarLinea(lado/2);
    dibujarRecursivo(lado/2, nivel-1);
    dibujarEsquina(lado);
    dibujarRecursivo(lado/2, nivel-1);
    dibujarEsquina(lado);
    dibujarRecursivo(lado/2, nivel-1);
    girar(M_PI);
    dibujarRecursivo(lado/2, nivel-1);
    girar(-M_PI);
    dibujarLinea(lado/2);
    salir();
    ponerColor(nivel+1);
  }
}
```

Hemos hecho uso de los siguientes pocedimientos auxiliares:

```
void entrar() {
  girar(2*M_PI/3);
}
```

```
void salir() {
  girar(2*M_PI/3);
}
void dibujarEsquina(double const lado) {
  dibujarLinea(lado/2);
  girar(-2*M_PI/3);
  dibujarLinea(lado/2);
}
void ponerColor(int const nivel) {
  Colores const colores[] = {rojo, amarillo, azul, verde};
  ponerColor(colores[nivel%4]);
}
```

Y ya sólo falta el programa que dibuja el exterior del triángulo:

```
void dibujarTriangulos(double const tam, int const nivel) {
  double tamanyo = tam;
  ponerCoordenadas(tamanyo*.1, tamanyo*.2);
  tamanyo = tamanyo*0.8;
  ponerColor(nivel+1);
  dibujarLinea(tamanyo/2);
  dibujarRecursivo(tamanyo, nivel);
  ponerColor(nivel+1);
  dibujarLinea(tamanyo/2);
  girar(2*PI/3);
  dibujarLinea(tamanyo);
  girar(2*PI/3);
  ponerColor(nivel+1);
  dibujarLinea(tamanyo);
}
```

3.26 Cálculo puntual de la matriz de mediotono de Judice-Jarvis-Ninke

123 130

Para encontrar el elemento de la posición (i, j) hay que vagar por la matriz, empezando en el nivel más externo y pasando a la submatriz adecuada del nivel anterior. Veamos cómo sería el recorrido para buscar el elemento $(4, 3)$ en una matriz de 8×8, que llamaremos L_3. Como $(4, 3)$ cae en la submatriz de 4×4 de la esquina superior izquierda (L_2) pasamos a buscarlo ahí. En este nuevo nivel, $(4, 3)$ cae en la submatriz de 2×2 de la esquina inferior derecha (L_1). A continuación deberíamos buscar la posición $(2, 1)$ en L_1. Observa que esta posición es la misma que la $(4, 3)$ de L_2, y que además cae en la esquina inferior izquierda de L_1 (L_0). La posición que debemos buscar en L_0 es la $(1, 1)$: sólo disminuye el índice de fila.

Una vez que sabemos cómo localizar una cierta posición, calcular el valor que ha de contener es relativamente sencillo. Cada una de las dos posibles implementaciones refleja una forma de realizar este cálculo. Con recursión, los cálculos se realizan cuando se hace el camino de vuelta; mientras que pasar a una de las submatrices se convierte en una llamada recursiva (la mejor forma de calcular 2^m en C++ es con `1 << m`):

```
int limb(int const i, int const j, int const n) {
  if (n == 0) {
    return 0;
  } else {
```

```
    int const m = 1 << (n-1);  // 2^(n-1)
    if (i <= m) {
      if (j <= m) return 4*limb(i, j, n-1);
      else return 4*limb(i, j - m, n-1) + 2;
    } else {
      if (j <= m) return 4*limb(i - m, j, n-1) + 3;
      else return 4*limb(i - m, j - m, n-1) + 1;
    }
  }
}
```

En la versión iterativa hay que hacer los cálculos según vamos pasando a las matrices más internas. No hay camino de vuelta; cuando localicemos la posición tendremos almacenado su valor en la variable `r`. Cada paso de nuestro camino exige aumentar el valor de `r` con 0, 1, 2 o 3, dependiendo de a qué submatriz pasemos. Como hay que tener presente el producto por 4, resulta que los incrementos anteriores sólo sirven para el primer paso, mientras que en el segundo hay que multiplicarlos por 4, en el tercero por 16 y así sucesivamente. La potencia de 4 con la que estemos trabajando se tendrá en la variable `b` y se incrementará en cada paso de nuestro camino:

```
int limb(int const i, int const j, int const n) {
  int pi = i, pj = j;
  int m = 1 << n;  // 2^n
  int r = 0, b = 1;
  while (m > 1) {
    m >>= 1;
    if (pi > m) {
      pi -= m;
      if (pj > m) {
        pj -= m;
        r += b;
      } else {
        r += 3*b;
      }
    } else if (pj > m) {
      pj -= m;
      r += 2*b;
    }
    b *= 4;
  }
  return r;
}
```

Las dos soluciones anteriores recorrían la matriz pasando a submatrices cada vez más pequeñas. También se puede recorrer en el sentido opuesto, empezando por una matriz L_0 y pasando a matrices cada vez más grandes. Si suponemos que los índices i y j son naturales entre 0 y $2^n - 1$, su representación en base 2 indica la posición relativa de un nivel con respecto al siguiente. En el ejemplo que pusimos al principio, las representaciones en base 2 de $4 - 1$ y $3 - 1$ son 11 y 10 respectivamente. La relación entre L_0 y L_1 viene definida por los bits menos significativos: L_0 está en la parte inferior de L_1 por el 1 de 11, y en la parte izquierda por el 0 de 10. Igualmente, los dos siguientes bits indican la relación entre L_1 y L_2: como ambos son 1, L_1 está en la esquina inferior derecha de L_2. Finalmente, la relación entre L_2

y L_3 viene dada por los dos siguientes bits que no aparecen porque son ceros a la izquierda y, por tanto, L_2 está en la esquina superior izquierda de L_3. El programa que sigue utiliza esta búsqueda. El acceso a los bits se hace explorando el bit menos significativo (con el resto de la división entre 2) y descartándolo a continuación (con una división entera entre 2):

```
int limb(int const i, int const j, int const n) {
  int pi = i-1, pj = j-1, k = n;
  int r = 0;
  while (k > 0) {
    if (pi % 2 == 1) {
      if (pj % 2 == 1) r = 4*r + 1;
      else r = 4*r + 3;
    } else {
      if (pj % 2 == 1) r = 4*r + 2;
      else r = 4*r;
    }
    k--;
    pi / 2;
    pj / 2;
  }
  return r;
}
```

3.27 Dibujo de árboles mediante fractales

124 130

Como se ha dicho en el enunciado, este programa es sencillo, al menos algorítmicamente:

```
void dibujarArbol(double const tamanyo, Posicion const& posicion,
                  double const angulo, int const nivel) {
  dibujarTronco(tamanyo, posicion, angulo);
  if (nivel > 1) {
    Posicion posAux;
    posAux = coordenada1(tamanyo, posicion, angulo);
    dibujarArbol(tamanyo/2, posAux, angulo-20, nivel-1);
    posAux = coordenada2(tamanyo, posicion, angulo);
    dibujarArbol(tamanyo/2, posAux, angulo-20, nivel-1);
    posAux = coordenada3(tamanyo, posicion, angulo);
    dibujarArbol(tamanyo/2, posAux, angulo, nivel-1);
    posAux = coordenada4(tamanyo, posicion, angulo);
    dibujarArbol(tamanyo/2, posAux, angulo+20, nivel-1);
    posAux = coordenada5(tamanyo, posicion, angulo);
    dibujarArbol(tamanyo/2, posAux, angulo+20, nivel-1);
  }
  else dibujarHoja(tamanyo, posicion, angulo);
}
```

Podemos observar que estamos usando un tipo nuevo llamado `Posicion`. Si bien no se verá hasta el capítulo 4, su uso es muy sencillo y simplifica la solución. Se podría haber hecho la solución sustituyendo las variables de tipo `Posicion` por dos variables de tipo `double`.

Pasemos a los detalles pendientes. En la cabecera vemos que hace falta indicar dónde empezamos a dibujar y el ángulo en que se trazará el árbol. Esto se ha adelantado por simplicidad, pero su necesidad va a ponerse de manifiesto en seguida.

Empecemos por ver cómo dibujar el tronco. Aparentemente, el tronco es un triángulo isósceles; pero la mitad superior del tronco *es* un subárbol, por lo que el tronco propiamente dicho será la *mitad inferior* de ese triángulo, esto es, un trapecio de forma que la base superior mide la mitad que la inferior, según el teorema de Tales (Tales de Mileto, 624–547 a.C.), y está centrada con respecto a ésta. El tamaño de la base es un tanto arbitrario; obviamente debe ser relativamente pequeña con respecto a la altura del árbol. Aquí hemos tomado 1/10 de la altura total del árbol. Entonces, si `tam == tamanyo/2`, las coordenadas del trapecio serán $(0, 0)$, $(0.05 \times \texttt{tam}, \texttt{tam})$, $(0.15 \times \texttt{tam}, \texttt{tam})$ y $(0.2 \times \texttt{tam}, 0)$. Pero hay que tener en cuenta que estas coordenadas son relativas a la base del dibujo (`posicion`) y están rotadas un `angulo`. Para calcular ese cambio de coordenadas se usa el procedimiento `cambioCoordenadas`.

```
void dibujarTronco(double const tamanyo, Posicion const& posicion,
                   double const angulo) {
  ponerColor(1.0, 0.7, 0.0);
  Posicion pos[4];
  double tam = tamanyo/2;       // altura
  pos[0].x = posicion.x; pos[0].y = posicion.y;
  pos[1] = cambiarCoordenadas(posicion, angulo, 0.05*tam, tam);
  pos[2] = cambiarCoordenadas(posicion, angulo, 0.15*tam, tam);
  pos[3] = cambiarCoordenadas(posicion, angulo, 0.2*tam, 0);
  double cx[4], cy[4];
  for (int i = 0; i < 4; i++) {
    cx[i] = pos[i].x; cy[i] = pos[i].y;
  }
  dibujarPoligono(cx, cy, 4);
}
```

Vamos con el trazado de las hojas. El `tamanyo` de la hoja lo establecemos como 1/10 del tronco, pero cuando estamos en un nivel bastante avanzado en la recursión ese `tamanyo` resulta demasiado pequeño y no se aprecia, por lo que establecemos un `tamanyo` mínimo. La posición debe ser encima del tronco, sobre su eje:

```
void dibujarHoja(double const tamanyo, Posicion const& posicion,
                 double const angulo) {
  ponerColor(verde);
  double tam = tamanyo/2; // tamaño del tronco
  double radio = tam*0.1; // un 1/10 del tamaño del tronco
  radio = radio < 10 ?  10 :  radio; // el tamaño mínimo es 10
  Posicion aux = cambiarCoordenadas(posicion, angulo, tam*0.1, tam*1.08);
  dibujarDisco(aux.x, aux.y, radio);
}
```

Los procedimientos para calcular las coordenadas dependen del sitio exacto donde deseemos colocar los subárboles, del `tamanyo` del árbol y del `angulo` actual de dibujo. Por ejemplo, la base del primer subárbol está elevada 1/5 del tamaño del árbol con respecto a su base. Si tenemos en cuenta que el lado superior del trapecio que forma el tronco es la mitad que el inferior y éste es, a su vez, 1/10 de la altura del árbol, la coordenada x de la base debe estar desplazada $0.05 \times$ `tamanyo` (otra vez Tales). Así, el procedimiento para la primera coordenada será:

```
Posicion coordenada1(double const tamanyo, Posicion const& pos,
                     double const angulo) {
  return cambiarCoordenadas(pos, angulo, tamanyo*0.05, tamanyo*0.2);
}
```

Los demás serán similares. Poco hay que decir sobre el procedimiento de cambio de coordenadas, que se resuelve con trigonometría elemental. Simplemente hay que observar que hemos tratado los ángulos en grados, mientras que las funciones trigonométricas predefinidas trabajan con radianes, por lo que se requiere la oportuna conversión:

```
Posicion cambiarCoordenadas(Posicion const& pos, double const angulo,
                            double const x, double const y) {
  Posicion aux;
  double anguloRadianes = PI*angulo/180;
  aux.x = pos.x + x*cos(anguloRadianes) - y*sin(anguloRadianes);
  aux.y = pos.y + y*cos(anguloRadianes) + x*sin(anguloRadianes);
  return aux;
}
```

Aún podemos mejorar el trazado de árboles descrito introduciendo cierta aleatoriedad en el dibujo de los subárboles. Para ello, podemos añadir un parámetro `corte` que indica la probabilidad con la que queremos dibujar los subárboles. Hemos de tener en cuenta que el tercer árbol se debe dibujar en cualquier caso. Para decidir si dibujamos cada subárbol, podemos usar una distribución de Bernoulli con el parámetro deseado (véase el ejercicio 3.13). Además, cada vez que hacemos una llamada recursiva, la probabilidad debe disminuir multiplicando por `indiceDisminucionProbabilidad` que está definida como una constante entre 0 y 1. Es necesario hacer pocas modificaciones al programa anterior: en primer lugar, añadir el parámetro `corte`, (que recoge la probabilidad con que se dibujarán los subárboles) en la cabecera de la función:

```
void dibujarArbol(double const tamanyo, Posicion const& posicion,
                  double const angulo, int const nivel, double const corte);
```

Y cada llamada recursiva (salvo la tercera) ha de protegerse dentro de un condicional como éste:

```
if (uniforme(0, 1) <= corte) {
  dibujarArbol(tamanyo/2, posAux, angulo-20, nivel-1,
               corte*indiceDisminucionProbabilidad);
}
```

3.28 Dragones y teselas

126

Observando la figura 3.6, podemos comprobar que los siguientes pasos generan una curva dragón:

1. Girar $\pi/4$ radianes (45 grados) a la derecha.
2. Generar un dragón a la derecha (según la línea previa de puntos).
3. Girar $\pi/2$ radianes (90 grados) a la izquierda.
4. Generar un dragón a la izquierda.

El trazador que nos dan para la realización de este ejercicio tiene, en cada momento, una posición de dibujo y una orientación. Si empezamos a dibujar en una posición y una orientación, debemos acabar

dibujando en una posición incrementada en el tamaño del dragón según la dirección original y en esa misma orientación; por tanto antes de acabar será necesario recuperar la orientación:

5. Girar $\pi/4$ radianes (45 grados) a la izquierda.

Pero lo descrito expresa el *caso inductivo*: todavía no hemos dibujado nada. Los trazos del dibujo se harán en el caso base, que consistirá únicamente en una línea (dragón de grado 0) de la longitud requerida. Los giros comentados siempre son relativos a la orientación en la que deseamos hacer el dibujo. En lo descrito estamos suponiendo que queremos dibujar un dragón "a la derecha" (tal y como se ve en la figura 3.6). Pero si queremos dibujar un dragón "a la izquierda", habrá que cambiar de mano. Para representar la orientación definimos sendas constantes:

```
int const derecha = 1;
int const izquierda = -1;
double const reduccion = sqrt(2)*0.5; // proporción del lado de dos dragones consecutivos
```

Así, el procedimiento recursivo para dibujar el dragón queda como sigue:

```
void dibujaDragon(double const tamanyo, int const orientacion, int const nivel) {
  if (nivel > 0) {
    girar(orientacion * (-PI/4));
    dibujaDragon(tamanyo*reduccion, derecha, nivel-1);
    girar(orientacion * (PI/2));
    dibujaDragon(tamanyo*reduccion, izquierda, nivel-1);
    girar(orientacion * (-PI/4));
  } else {
    dibujarLinea(tamanyo);
  }
}
```

Vamos a hacer un procedimiento que dibuje cuatro dragones acoplados tal y como aparecen en la figura 3.7:

```
void dibujaDragones(double const tamanyo, int const nivel) {
  double angulo = PI/4;
  Colores const color[] = {rojo, azul, verde, amarillo};
  for (int i = 0; i < 4; i++) {
    ponerCoordenadas(tamanyo*0.5, tamanyo*0.5);
    ponerAngulo(angulo);
    ponerColor(color[i]);
    dibujaDragon(tamanyo*0.4, derecha, nivel);
    angulo += PI/2;
  }
}
```